MICROWARE®

# A/BASIC COMPILER

## VERSION V2.1S

## REFERENCE MANUAL

Microware Systems Corporation
5835 Grand Avenue
Box 4865
Des Moines, Iowa 50304
515-279-8844

A/BASIC V2.1S COMPILER REFERENCE MANUAL

Copyright 1978 Microware Systems Corporation All Rights Reserved

Microware Systems Corporation distributes this manual for use by its licensees and customers. The information and programs contained herein is the property of Microware Systems Corporation and may not be reproduced or duplicated by any means without express written authorization.

---

SOFTWARE LICENSE AND LIMITED WARRANTY:

A/BASIC is copyrighted property of Microware Systems Corporation and is furnished to its customers for use on a single computer system owned by the customer. The program may not be copied in any form except for use on the customer's computer system. This license is not transferable and the customer may not make this software available to any third party without express written consent of Microware Systems Corporation.

Purchase of this program and manual is considered implied consent of the provisions of the software license warranty.

Microware Systems Corporation warrants this software to be free from defects for a period of 90 days after date of purchase. Any corrections may be made by means of printed or magnetic media at the option of Microware Systems Corporation.

Microware Systems Corporation reserves the right to make changes without notice (except within the warranty period) in the interest of product improvement at any time. Microware Systems Corporation cannot be responsible for any damages, including indirect or consequential, caused by reliance on this product's performance or accuracy of its documentation.

---

MICROWARE SYSTEMS CORPORATION
P.O. BOX 4865
5835 GRAND AVENUE
DES MOINES, IOWA 50312
(515) 279-8844

FIRST EDITION

PART NUMBER BASD-M

Printed in U.S.A.

INTRODUCTION 1
A/BASIC PROGRAM STRUCTER 2
ARITHMETIC OPERATIONS 3
NUMERIC VARIABLES 4
ARITHMETIC OPERATORS ———— 5
ARITHMETIC FUNCTIONS 6
ARITHMETIC ERRORS 6
MULTIPLE PRECISION ARITHMETIC 7
STRING OPERATORS 8
STRING CONCATENATION ———— 9
STRING FUNCTIONS 10
STRING OPERATIONS ON THE I/O BUFFER 12
STRING EXPRESSIONS 13

COMPILER DIRECTIVE STATEMENTS

ORG & BASE 14
DIM 16
DECLARING SIMPLE VARIABLES 16
OPTION 18
REM ———— 18
END 19

ASSIGNMENT STATEMENTS

ARITHMETIC ASSIGNMENT ———— 20
POKE ASSIGNMENT 20
STRING ASSIGNMENT 20

CONTROL STATEMENTS

CALL STATEMENT 21
FOR/NEXT STATEMENT ———— 21
22
GOSUB/RETURN STATEMENT 22
IF/THEN STATEMENT 23
ON ERROR GOTO STATEMENT
ON GOTO/ON GOSUB STATEMENT 24
STOP ———— 24

REAL-TIME AND SYSTEM CONTROL STATEMENTS

GEN Statement 25
IRQ ON/OFF Statements 25
ON Interrupt Statements 25
RETI Statement 26
STACK Assignment Statement 26
SWITCH Statement 26
TASK ON/OFF Statements 27

INPUT/OUTPUT STATEMENTS

INPUT Statement 28
PRINT Statement 29

A/BASIC DISK I/O OPERATIONS

Disk I/O Conventions 30
OPEN FILES and CLOSE FILES Statements 31
OPEN Statement 31
CLOSE Statement 32
WRITE Statement 33
READ Statement 34
CHAIN Statement 3~~2~~5
RESTORE and SCRATCH Statements 36
KILL Statement 36
Disk Functions 37

USER-DEFINED STATEMENTS

User-Defined Statements 38

COMPILATION PROCEDURES

Compilation Procedures 40
Error Messages and Processing 41

APPENDIX A - MEMORY UTILIZATION

Memory Map - Compile Time A1
Compiler Internal Memory Addresses A2
A/BASIC Run-Time Environment A3
APPENDIX B - A/BASIC LANGUAGE SUMMARY B1
APPENDIX C - DIFFERENCES BETWEEN VERSIONS 1.0 AND 2.0 C1

APPENDIX D - A/BASIC ERROR CODES

Compile-Time Error Codes D1
Run-Time Error Codes D2

# APPENDIX E - A/BASIC GLOSSARY

Additional information on your specific version of A/BASIC is listed on the diskette supplied on a file called INFO.

## INTRODUCTION

A/BASIC is an optimizing two-pass BASIC compiler for the M6800 family of microprocessors which converts programs written in BASIC to pure 6800 machine language.

A/BASIC is oriented towards applications previously programmed in assembly language due to its capablity to product extremely fast compact object programs.

The compiler's output can be run as a stand-alone RAM, ROM or PROM based program which may be run without any run-time package. A built-in linker/editor automatically selects subroutines from A/BASIC's internal library and inserts one and only one copy of subroutines required directly into the object program.

A/BASIC V2.0 requires a minimum of 12K bytes of RAM and the host disk operating system to compile programs. If the computer utilizes the RT/68 operating system several real-time and multiprogramming capabilites may be used from BASIC source programs.

Depending on the specific program, A/BASIC can produce programs which may reflect a 50 to 1000 times speed improvement over interpreters with a significant memory savings - typically 25-50% less memory required.

A/BASIC V2.0 also contains statements for creating, manipulating and performing I/O to disk data files which make it suitable for a wide range of system programmming applications.

## A/BASIC PROGRAM STRUCTURE

An A/BASIC programs consists of a series of source lines. A source line may optionally begin with a line number, which is then followed by one or more A/BASIC statements. If the source line contains more than one statement a colon : character is used to separate the statements. A source line may contain up to 80 characters.

Line numbers are decimal numbers which are up to four digits and positive. These must appear sequentially in a program and may not be duplicated.

Spaces in A/BASIC statements are not required however they may be used to improve readability (except when used in string constants). Unlike interpreters, REMark statements do not affect program size and may be used generously.

The last statement of a program is an END statement and processing ceases when END is read.

Example of program structure:

```
100 PRINT "THIS PROGRAMS FINDS THE AVERAGE OF A SERIES OF NUMBERS"
PRINT "HOW MAY NUMBERS " : INPUT N :T=0
FORX=1TON : PRINT "NEXT NUMBER" : INPUT I : T=T+I : NEXT X
PRINT:PRINT: PRINT "AVERAGE IS";T/N
PRINT "DO YOU WANT TO CONTINUE" : INPUT A$
200 IF A$="YES" THEN 100
STOP : END
```

When a program such as the one above is compiled, the compiler's listing will automatically tab and format it for better readability.

## ARITHMETIC OPERATIONS

### NUMBERS

A/BASIC's numeric data type is internally represented as 16 bit (2 byte) two's compliment integers. This permits a equivalent decimal range of +32767 to -32768. This data representation is quite natural to the 6800's machine instruction set which allows A/BASIC to produce extremely fast and compact machine code.

Because the compiler supports boolean operations, unsigned 16 bit binary numbers amy also be used for many functions. The range for these are: 0 to +65535. These numbers are used for referencing memory addresses in many cases.

A/BASIC programs may include numeric constants in either decimal or hexadecimal notation. In the latter case a dollar sign must precede to hex value. Either type may have a preceding minus sign to represent a negative value or a pound sign # to represent the logical compliment (1's compliment or boolean NOT).

Examples of legal number constants:

```
200   -5000   $0100   -$3000  12345  #1  #$5000   $FFF0   #$C0F1
```

Examples of ILLEGAL NUMBERS:

9.99 (fractions not allowed)
1000000 (number is too large)
+20 (plus sign not allowed - positive assumed if not minus)

Because of the way binary numbers are represented as either unsigned or 2's compliment as well as the differences between hex and decimal notation of identical numbers, all the following number constants have a binary value which are the same:

$-1_{10}$     $FFFF   #0    65535

## NUMERIC VARIABLES

Legal numeric variable names in A/BASIC consist of a single letter A-Z or a single letter and a digit 0-9. The following are legal variable names:

X N R2 Z9 A0 P1

If declared in a DIM statement, numeric variables may be arrays of one or two dimensions. The maximum subscript size is 255, therefore the largest one-dimensional array has 255 elements and the largest two-dimensional array has 255*255 = 65025 elements (which is too big to actually exist within the 6800 memory space). Subscripts start with 1.

When referencing subscripted variables the subscripts may be numeric constants, variables or expressions as long as the evaluated result is a positive number from 1 to 255. A/BASIC does not perform run-time subscript checking for overrange errors which would cost considerably in terms of program size and speed.

References to two-dimensional array require the program to perform a multiplication to calculate the actual element address. Even though A/BASIC uses a special fast 8 by 8 bit multiply for array address calculations it takes about 250 MPU cycles minimum to access a two-dimensional element as opposed to about 20 MPU cycles for a one-dimensional access.

Examples of legal subscripting:

```
N(M)  A(12)  X2(C)   Z4(N,M)    H(N*(A/B),A+2)    R4(N*N+M)
```

A/BASIC considers a simple variable with the same name as an array to be the first element of an array. For example if there is a two-dimension array A(20,40) using the variable name A without any subscript is equivalent to using A(1,1).

Each numeric variable or element of an array is assigned two bytes of RAM for storage.

## ARITHMETIC OPERATORS

The five legal operators for arithmetic are:

| | |
|---|---|
| + | ADD |
| - | SUBTRACT |
| * | MULTIPLY |
| / | DIVIDE |
| - | NEGATIVE (UNARY) |

There are also four boolean operators:

| | |
|---|---|
| & | AND |
| ! | OR |
| % | EXCLUSIVE OR |
| # | COMPLIMENT (UNARY) |

All the above operators may be mixed in arithmetic expressions. The boolean operators operate in a bit-by-bit manner across all 16 bits of the numeric value.

The order of operation determines in which order A/BASIC processes expressions. The compiler will convert arithmetic expressions to an internal form during compilations and rearrange expressions following the order of operations so it may produce machine instructions which are shortest and fastest. Expressions are evaluated in the following order:

1. FUNCTIONS
2. UNARY NEGATIVE AND NOT
3. AND, OR, EXCLUSIVE OR
4. MULTIPLY, DIVIDE
5. ADD, SUBTRACT

Parentheses may be used to alter the normal order of evaluation where required. Some legal usage of expressions:

```
A*B(N,M+4)  $200+Z   A&B!C*D/F+(H+(J*2)&$FF00)  N+A(Z)/VAL("FOUR")
```

## ARITHMETIC FUNCTIONS

A/BASIC supports the following functions:

| | |
|---|---|
| ABS(expr) | The absolute value of the argument. |
| CLK | Current value of the RT/68 real time clock in ticks. |
| RND or<br>RND(expr) | Next number from the random sequence. The number will be in the range 0 to +32767. If an argument is supplied, it is evaluated and used to "seed" the random number generator (randomize it). If the RT/68 operating system is used and the real time clock is active, RND(CLK) will produce an extremely random (mathemetically) sequence. |
| PEEK(expr) | Used to access a byte value at an address determined by the result of the argument. |
| POS | The current character position in the output buffer (print position). |
| SWAP(expr) | Byte swap of the result of the argument. |
| ERR | Returns error type of most recent error condition. |

## ARITHMETIC ERRORS

Arithmetic operations may produce several types of errors which may be detected and processed. Addition and subtraction may result in a carry or borrow. Either one will result in the C bit of the MPU's condition code register being set. The ON OVR GOTO and ON NOVR GOTO statements may be used to detect this. This also permits addition and subtraction in larger representation than 16 bits. (See MULTIPLE PRECISION ARITHMETIC)

Multiplication of two 16 bit numbers may result in a product of up to four bytes long. A/BASIC will detect this error (see ON ERROR GOTO) and preserve the high-order 16 bits of the correct 2's compliment result at addresses $002B and $002C.

Division attempted with a divisor of zero will also produce an error which is detected at run-time with the ON ERROR GOTO statement.

## MULTIPLE PRECISION ARITHMETIC

Sometimes it is necessary to deal with numbers larger than the basic 2-byte A/BASIC representation. A/BASIC allows addition and subtraction of numbers of multiples of 16 bits by means of the ON OVR GOTO and ON NOVER GOTO statements. OVR means overflow (carry or borrow as represented by the MPU C bit) and NOVR means NOT OVERFLOW.

The example below shows addition and subtraction of 32-bit integers using the convertion that two variables are used to store each number: A1 and A2 are the first number with A1 being the most significant bytes; and B1 and B2 used similarly. To add A1-A2 to B1-B2 the following subroutine may be used:

```
100 A2=A2+B2 : ON NOVR GOTO 200 : REM ADD L.S. BYTES
    A1=A1+1  : REM ADD 1 TO MS BYTES FOR CARRY
200 A1=A1+B1 : REM ADD MS BYTES
    RETURN
```

To subtract B1-B2 from A1-A2 a similar subroutine may be used:

```
100 A2=A2-B2 : ON OVER GOTO 300 : REM SUB. LS BYTES
200 A1=A1-B1 : RETURN : REM SUB MS BYTES
300 GOSUB 200 : A1=A1-1 : RETURN : REM BORROW CASE
```

For cases where multiply, divide or even floating-point arithmetic must be used, external subroutines may be used. In such cases several compiler features and capablities may be use to simplify the interface.

1) Use the CALL statement to call the subroutines.
2) Set up conventions so values are passed to the external subroutines in certain memory addresses that have been assigned A/BASIC variable names so the A/BASIC program may easily manipulate them.
3) Use A/BASIC's string processing capabilites to full advantage in handling I/O and storage of numeric values. Floating point numbers can be passed as strings in ASCII format.

## STRING OPERATIONS

A/BASIC features a complete set of string processing capabilities which allow BASIC programs to perform operations on character-oriented data. Character-type data is represented in A/BASIC in string form which is defined as variable-length sequences of characters.

### String Literals:

A string literal or constant consists of a series of characters enclosed in quotation marks:

"THIS IS A STRING LITERAL"

Any characters may be included in a string literal except for the ASCII characters for carriage return, null or SUB ($1A - used for end-of-file on source programs). A string literal may include up to as many characters as may fit in an A/BASIC source line. The quotes are not considered a part of the string. If a quote is to be included as part of the string two are used so the literal:

"AN EMBEDDED "" QUOTE"

is interpreted to mean the constant string:

AN EMBEDDED " QUOTE

String literals are used in string assignment statements or expressions, and in PRINT or WRITE statements.

### String Variables:

A/BASIC allows string vaiables which may be either single strings or arrays of strings. String variable names consist of a single letter A-Z followed by a dollar sign such as A$, M$ or Z$.

String variables may be used with or without explicit declaration. If a string variable is encountered for the first time in the source program as it is being compiled without having been previously declared in a DIM statement the compiler will assign 32 bytes of storage for the string. This is the maximum number number of characters that may be assigned to the variable. If the assignment statement produces a result which has more characters than assigned for the variable the first N characters will be stored where N is the length of the variable storage assigned.

A string variable or array may be declared to have a size of 1 to 255 characters in length if the string is declared by a DIM statement before it is used (see DIM statement description).

If the string name is declared as an array, the maximum subscript size is 255. Legal usage of string arrays require that only one subscript (which may be an expression) be used:

```
    A$(5)    N$(X+5)    X$(A+(N/2))
```

## String Concatenation

The string concatenation operator + is used to join strings to form a new string or string expression. For example:

```
    "NEW "+"STRING"
```

produces the value: "NEW STRING".

## Null Strings

Strings which have no characters are represented as literal as "" which represents an empty string. This is typically the initial value assigned to a string which is to be "built up". The string assignment statement:

`A$=""`

is somewhat analogous to the arithmetic assignment A=0 in the sense that both cause a variable to be assigned a defined value of "nothing". This is important because before a string variable is used in a program it has a value which is random and meaningless.

## String Functions

A/BASIC includes may functions which manipulate strings or convert strings to/from other types. Some of the functions which include $ in their name produce results which are of a string type and may be used in string expressions. In the description of string functions that follow, the notation:

N refers to a numeric-type argument which is a constant, variable or expression

X$ refers to an argument of string type which may be a string literal, variable or expression.

The following functions produce STRING results:

CHR$(N) returns a character which is the value of the number N in ASCII.

LEFT$(X$,N) returns the N leftmost characters of X$. For example, the function LEFT$("EXAMPLE",3) returns "EXA".

MID$(X$,N,M) returns a string which is that part of X$ beginning with its Nth character and extending for M characters. For example: the function MID$("EXAMPLE",3,4) returns "AMPL".

RIGHT$(X$,N) returns the N rightmost characters of X$. An example this function is RIGHT$("EXAMPLE",3) which produces "PLE".

STR$(N) is a function used to convert a number from numeric type to string type which is a string of characters which are decimal digits. For example STR$(1234) returns the string "1234". This function has the inverse effect of the VAL function.

TRM$(X$) is a function which removes trailing blanks from a string and is typically used after a string is read from input. For example TRM$("EXAMPLE  ") returns "EXAMPLE".

Note on the above functions: if there are not enough characters in the argument to produce a full result, the characters returned will be those processed until the function "ran out" of input, or a null string, whichever is appropriate. The STR$(N) function will result in a run-time error detectable by the ON ERROR GOTO function if its argument is not legal or convertible to a string.

The following functions have string argument(s) and produce a result which is of type numeric:

ASC(X$) returns a number which is the ASCII value of the first character of the string. For example ASC("EXAMPLE") returns a value of $45 or decimal 53 which is the ASCII code for the character E. This is the inverse function of CHR$.

LEN(X$) returns the length of the string. LEN("EXAMPLE") returns a value of 7. LEN("") returns a value of 0.

SUBSTR(X$,Y$) is a substring search function which searches for the string X$ in the string Y$. If an identical substring is found the function will return a number which is the position of the first character of the substring in the target string. If the substring is not found the function returns a value of 0. For example, the function SUBSTR("EXAMPLE","PL") returns a value of 5. SUBSTR("EXAMPLE","NOT") returns a value of 0.

VAL(X$) converts a string of characters for decimal digits and optionally a leading minus sign to a numeric value. This has the inverse effect of STR$. If the string arguement is not a legal conversion string (it has too many, non-decimal or no digit characters) a run-time error detectable by ON ERROR GOTO occurs. For example: VAL("1234") returns a numeric value of 1234. VAL("THREE") results in an error.

## String Operations on the I/O Buffer

Commonly BASICs have limitations because of the input formatting when reading mixed data types. For example BASIC input conventions cause commas which are part of the input data to break up what are really one long string, etc. A/BASIC has a special string variable, BUF$ which is defined to be the contents of the run-time I/O buffer which may be used as any other string variable. BUF$ is 129 bytes long.

The following I/O statement forms are legal for filling or dumping the I/O buffer when used with BUF$:

```
INPUT BUF$        PRINT BUF$
READ #N,BUF$      WRITE #N,BUF$
```

Example of usage of BUF$ as a variable:

```
BUF$=MID$(BUF$+A$,N,M)
```

## String Expressions

String expressions may be created using string variable names, the concatenation operator and string functions. Expressions are evaluated from left to right and the only precedence of operations involved is evaluation of function arguments performed before concatenation.

At run-time, string operations are performed on data moved to the string buffer, a compiler-allocated area normally 255 bytes long. Because this is always the last data storage allocated by the compiler, any memory available beyond this area may be used to allow automatic buffer exapansion if operations on extremely complex string expressions are involved, or if string variables or constants have a long length.

Examples of legal string expressions:

```
"CAT"
A$
A$+"DOG"
LEFT$(B$,N)
A$+RIGHT$(D$,Z)+"TH"
MID$(A$+B$,N,LEN(A$)-1)
"AA"+LEFT$(RIGHT$(TRM$(A$)+B$,Z,4),X+2)+C$
```

## COMPILER DIRECTIVE STATEMENTS

### ORG and BASE Statements

SYNTAX: ORG addr
        BASE addr

These statement types are used to control how A/BASIC assigns memory in the object program. The ORG statement is used to assign starting addresses for the object code and the BASE statement is used to define the addresses used for variable storage.

Both statement types may be used as often as desired so memory assignments for program and data storage may be segmented as desired.

A/BASIC uses two internal "pointers" that control how run-time memory is allocated. The "object code pointer" always maintains the address where the next instructions generated by the compiler will be stored. The ORG statement assigns a value to this pointer. When A/BASIC is first entered, a default value of $1000 is assigned to the pointer so unless an ORG statement is processed before the first executable BASIC statement, the program's default starting address is $1000.

For example, the statement:

```
    ORG=$2400
```

will cause instructions generated for following BASIC statements to begin at address $2400. The ORG statement may be used to create "modules" at different addresses within a single program.

The BASE statement is also used to control memory assignment in a similar manner but it applies to allocation of RAM for variable storage. An internal "data address pointer" is maintained by A/BASIC to hold the next address available (at run-time) for variable or temporary storage. It is initialized by default to address $0030.

A/BASIC assigns RAM corresponding to BASIC variables the first time they are encountered in the source program at compilation time. When a "new" variable name is encountered, A/BASIC assigns the variable run-time storage corresponding to the current value of the data address pointer

ORG and BASE Statements - Cont'd

which is then updated by increasing it by the size of the variable storage assigned. It therefore is again pointing to the next available RAM location.

An important function of the BASE statement is to allow specific memory assignments for specific variable names. Reasons for this application are:

1. To take advantage of the 6800's "direct page" addressing mode. When commonly used variables are located in the page of memory from $0000 to $00FF the compiler uses the direct addressing mode which can reduce program size and increase execution speed substantially.

2. To assign RAM consistant with actual RAM addresses that are available in the computer the software is to run on.

3. To assign specific variable names and types with memory addresses which have special functions. For example addresses of ACIAs, PIAs or other interface registers may be given BASIC variable names. A common type of "trick" is to declare the memory used by a video display (memory-mapped) as a BASIC string array which permits fast, simple updates to the video image.

When using the BASE and ORG statements the programmer must take care to ensure there are no conflicts between program and data storage by using assignments which are not overlapped.

Sometimes it is useful to declare a variable without generating code at the time it is declared. If the variable is an array, the DIM statement may be used. If it is a simple type, the DIM statement declaration with a size of one may be used for a declaration. For example, to assign the address $8010 to the variable P1 the following sequence may be used (assuming P1 had not been referenced previously in the program):

```
BASE=$8010
DIM P1(1)
```

## DIM Statement

This statement type is used to declare arrays and optionally, other simple variables. Arrays must be declared in a DIM statement before they are referenced in the program. The DIM statement may be used to declare more than one array. Arrays may not be redefined in following DIM statements. Array subscripts have a legal range of 1 to 255.

### Numeric Arrays

Numeric arrays may be declared to have one or two dimensions. Two dimensional arrays are stored in row-major order. Each element of a numeric array requires two bytes of storage. Examples of numeric array declaration:

```
DIM B(20),C(10,20),D($10,$20)
```

### String Arrays

String arrays may only be one-dimensional, however, the DIM statement is also used to specify the string size (1 to 255 characters) so the declaration for a one dimensional string array will have two subscripts: the number of strings and the length of each string. A single string may be declared in the DIM statement with a length specification only. Examples:

```
DIM A$(80)      one string of 80 characters
DIM B$(16,72)   16 strings of 72 characters
```

In the two examples above, A$ is used in the program WITHOUT any subscripts because it is <u>not</u> an array. B$ would be used in the program with <u>one</u> subscript because it is a one-dimensional array. For example:

```
A$=B$(N)
B$(X+2)=A$
```

## Declaring Simple Variables

Because A/BASIC allocates memory for variables as they are encountered for the first time, it is often useful to declare a variable name so it may be assigned storage at a particular point, but without generating code. This is often the case when it is desired to assign a variable a certain memory address. A/BASIC processes a variable declared as an array but used without subscripts in the program as the first element of the array be internally

## Declaring Simple Variables - Cont'd

assuming a subscript of (1) for a one dimensional array or (1,1) for a two-dimensional array. Because of this a declaration of a variable in a DIM statement with a subcript of 1 is legal but the variable may be used throughout the program without a subscript.

Example: Suppose a program is to be used to read from and write to an ACIA interface at address \$8008 - \$8009 and a PIA at addresses \$8020 - \$8023, and they are to be assigned variable names. A DIM statement at the beginning of the program may be used to assign variable names to these devices:

```
BASE=$8008     set compiler data pointer
DIM A(1)       declare ACIA as variable
BASE=$8020     reset data pointer
DIM P(1),Q(1)    declare PIA "A" and "B" registers
BASE=$0030     restore data pointer for other variables
```

The program may now refer to either the PIA or ACIA by variable name. To access the PIA "B" registers:

```
N=Q
```

or to read the ACIA:

```
N=A
```

## Option Statement

SYNTAX: OPT option, ... ,option

The OPT statement is used to turn on compiler options listed below:

I - Inhibit object file generation
H - Print instructions generated in hex form
N - No listing
S - Print symbol table

None of the above options are automatically selected by default. The H option provides the advanced programmer with the capability to examine the actual machine instructions generatedfor each BASIC source statement.

The S option causes the compiler's symbol table to be printed in hex at the end of the second pass. The symbol table data includes the symbol name, its run-time memory address, and its size. The two rightmost columns represent:

Row and Column size for numeric arrays
String length and #strings for string variables or arrays.

The symbol table dump may include symbols not used in the original source program which are compiler-allocated storage for special purposes. They are:

IO - Input/output buffer (129 bytes if used)
ST - String buffer (255 bytes plus additional available memory)
RR - Random number generator seed value (2 bytes)
ANY SYMBOL WITH A ASTERISK - For/next loop terminate/increment

Examples:
```
OPT S
OPT I,H,S
```

## REMARK STATEMENT

The REM statement is used to insert comments in the BASIC source program. The first three letters must be REM. On multiple statement lines the REM statement may only be used as the last statement on the line. This statement does not affect object program size or speed.

## END Statement

The END statement is the last statement of the source program. It causes compilation to cease and any statements following are ignored. If the BASIC source program omits an END statement and end-of-file is read on the source file END will be automatically assumed.

END does not result in generation of code. If it is desired to return to the operating system at run-time, a STOP statement must be used before the END.

## ASSIGNMENT STATEMENTS

### Arithmetic Assignment

```
SYNTAX:   LET var = expr
          var = expr
```

The expression is evaluated and the result is stored in the variable which may be an array. Use of the keyword LET is optional.

### POKE Assignment

```
SYNTAX:   POKE(expr) = expr
```

The expression is evaluated, and the result is truncated to a single byte value which is stored at the address determined by evaluation on the expression in parentheses. If the form POKE(number) is used to specify the address the fastest possible code is generated. The least significant byte of the result is stored.

### String Assignment

```
SYNTAX:   strvar = strexpr

          LET strvar = strexpr
```

The string expression is evaluated and the result assigned to the string variable specified, which may be an array element. If the result of the evaluation produces a result with a longer length than the size of the result variable, the first N characters only are stored where N is the length of the result variable.

## CONTROL STATEMENTS

### Call Statement

Syntax: CALL address

Description: The CALL statement is used to directly call a machine-language subroutine at the address specified. The subroutine will return to the BASIC program if it terminates with an RTS instruction and does not disturb the return address on the stack.

| | | |
|---|---|---|
| Examples: | CALL $EOCC | Call subroutine at address $EOCC |
| | CALL 1024 | Call subroutine at decimal addr. 1024 |

### For/Next Statement

Syntax: FOR var = expr TO expr STEP expr
NEXT var

Description: The FOR/NEXT uses a variable var as a counter while performing the loop delimited by the NEXT statement. If no step is specified, the increment value will be 1. The FOR/NEXT implementation in A/BASIC differs slightly from other BASIC due to a looping method that results in extremely fast execution and minimum length. Note the following characteristics of FOR/NEXT operation:

1. var must be a non-subscripted numeric variable.
2. The loop will be executed at least once regardless of the terminating value.
3. After termination of the loop, the counter value will be GREATER than the terminating value because the test and increment is at the bottom (NEXT) part of the loop.
4. FOR/NEXT loops may be exited and entered at will.
5. At compile time, up to 16 loops may be active, and all must be properly nested.
6. The initial, step, and terminating values may be positive or negative. The loop will terminate when the counter variable is greater than the terminating value.

Examples: FOR N = J+1 TO Z/4 STEP X*2
FOR A =-100 TO -10 STEP -2

## Gosub/Return Statements

Syntax: GOSUB line #
        RETURN

Description: The GOSUB statement call a subroutine starting at the line number specified. If no such line exists, an error message will be generated on the second pass. The machine stack is used for return address linkage. The RETURN statement terminates the subroutine and returns to the line following the calling GOSUB. Subroutines may have multiple entry and return points. The GOSUB and RETURN statements compile directly to JSR and RTS machine instructions, respectively.

## If/Then Statement

Syntax: IF <expr> <relation> <expr> THEN line #
        IF <expr> <relation> <expr> GOSUB line #

Description: The IF/THEN or IF/GOSUB is used to conditionally branch to another statement or conditionally call a subroutine based on a comparison of two expressions. Legal relations are:

| | |
|---|---|
| < | less than |
| > | greater than |
| = | equal to |
| <> | not equal to |
| <= =< | less than or equal to |
| >= => | greater than or equal to |

If the statement is an IF/GOSUB the subroutine specified will be called if the relation is true and will return to the statement following. Because of A/BASIC's multiple statement line capability, the IF statement can be used as an IF.. THEN.. ELSE function if another statement follows on the same line.

Examples:

```
If N = 100 THEN 1210
If A+B=C*D  GOSUB 5500
IF x>= 200 THEN 240  :  GOTO 1100
IF A$=B$ THEN 300
```

## ON ERROR GOTO Statement

SYNTAX: ON ERROR GOTO
ON ERROR GOTO line#

This statement provides a run-time error "trap" - the capability to transfer program control when an error occurs.

When an ON ERROR GOTO statement is executed the compiler saves the address of the line number specified in a temporary location. If any detectable error occurs during execution of following statements, the program will transfer to the line number given in the ON ERROR GOTO statement last executed. This would normally be the line number where an error recovery routine begins.

If the ON ERROR GOTO statement is used WITHOUT a line number specified, it has the effect of "turning off" the error trap - errors in following statements will be ignored.

After an error has been detected, the ERR function may be used to access a value which is an error code identifying the type of error which most recently occured. The exact error codes are related to the error codes used by the host operating system and are listed in the appendix.

The types of errors that can be detected by ON ERROR GOTO and the types of statements they occur in are listed below:

| | |
|---|---|
| DIVIDE BY ZERO | ARITHMETIC EXPRESSIONS |
| ASCII-TO-BINARY CONVERSION ERROR | INPUT, READ, VAL(X$) |
| MULTIPLY OVERFLOW | ARITHMETIC EXPRESSIONS |
| DISK ERRORS | DISK I/O |

Example of usage:

```
100 ON ERROR GOTO 500
120 INPUT A(N)
    N = N+1 : IF N=50 THEN 600 : GOTO 120
600 PRINT "ILLEGAL INPUT ERROR - RETYPE"
    GOTO 120
```

## On-Goto/on-Gosub Statements

Syntax: ON expr GOTO line # , line # , . . . , line #
ON expr GOSUB line # , line # , . . . , line #

Description: The expression is evaluated and one line number in the list corresponding to the value is selected for a branch or subroutine call, i.e., if the expression evaluates to 5, the fifth line number is used. If the result of the expression is less than specified, the next statement is executed.

Examples:
```
ON A*(B+C) GOTO 200,350,110,250,350
ON N GOSUB 500,510,520,500,100
```

## Stop Statement

Syntax: STOP

Description: The STOP statement is used to terminate execution of a program by causing a jump to the entry point pos. The END statement should not be confused with STOP as STOP will not terminate compilation of the program.

## REAL-TIME AND SYSTEM CONTROL STATEMENTS

### Gen Statement

Syntax: GEN number , number , ... , number

Description: The GEN statement allows data or machine language instructions to be directly inserted in the program. The list of values supplied are inserted directly into the object program. If a value given in the list is less than 255 one byte will be generated for that value regardless of leading zeros.

```
Example:  GEN $BD,$E141,$CE,1024  (PRODUCES 6 BYTES)
          GEN 0040,$00,32767  (PRODUCES 4 BYTES)
```

### IRQ On/Off Statements

```
Syntax:  IRQ ON
         IRQ OFF
```

Description: These statements are used to control the state of the MPU interrupt mask flag in the condition code register and are used to enable/disable interrupt recognition. These statements correspond directly to the 6800 CLI and SEI instructions. Refer to the RT/68 Systems Manual and the M6800 Programming guide for specifics on interrupt processing.

### On Interrupt Statements

```
Syntax:  ON IRQ GOTO  line #
         ON NMI GOTO  line #
```

Description: These statements are used for generating programs to be used in the RT/68 SINGLE TASK MODE (non-multiprogramming) where interrupts are processed as vectors to specific service routines. When encountered in a program these statements cause the absolute address of the program corresponding to the line number specified to be stored at the interrupt vector addresses in the operating system scratchpad ($A000 for IRQ and $A006 for NMI).

The line number specified should be the beginning of the interrupt service routine which would typically service the device causing the interrupt. This routine is similar to a BASIC subroutine except it is terminated by an RETI (return from interrupt) statement instead of a RETURN statement.

Examples: ON IRQ GOTO 2010
ON NMI GOTO 400

Interrupt Return Statement

Syntax: RETI

Description: This statement terminates an interrupt-caused routine by loading the MPU register contents prior to the interrupt from the machine stack and resuming program execution from the point where the interrupt was acknowledged. This statement corresponds directly to the machine language RTI instruction.

Stack Assignment Statement

Syntax: STACK = address

Description: This statement is used to initialize or change the MPU stack pointer register. This is typically one of the first statements in an A/BASIC program if used. If a STACK statement is not included in the program, the operating system scratchpad stack ($A049 to $A016) will be used by the program for the machine stack. This is adequate for most programs that do not: nest subroutines extensively; use interrupts; run in the multiprogramming mode; or process elaborate arithmetic expressions. Otherwise, a specific memory area should be dedicated for the stack and the STACK instruction used to load the stack pointer with the TOP (highest address) of the stack.

Example: STACK = $A042

Switch Statement

Syntax: SWITCH

Description: This statement is used in the RT/68 MULTITASK MODE to call the executive, thereby causing the task to be suspended while other tasks may run. The task selection/switching process is described in section 4 of the RT/68 Systems Manual. The machine instructions generated by this statement include operations that set the MPU interrupt mask and the RT/68 task switch flag byte (RELFLG) and a software interrupt (SWI) which activates the executive. When the task is run again, execution resumes at the statement following the SWITCH statement.

### Task On/Off Statements

Syntax: TASK <task #>ON
        TASK <task #>OFF

Description: When used in the RT/68 MULTITASK MODE this statement will find the task status byte of the specified task in the status table and either set or clear the task state bit. This causes the task to be either activated (able to run) or disactivated (may not run) when the executive is selecting a task to execute. Note that this statement does not directly cause the task to start/stop- it merely enables/disables it from consideration when the executive is searching for a task to run. A task may use the TASK N OFF to turn itself off and become dormant. This is often used when the task was activated by another to perform a specific function and has completed its operation.

## INPUT/OUTPUT STATEMENTS

All input and output statements use a 129-byte buffer for intermediate storage of data. This buffer may contain up to 128 characters. The buffer is automatically allocated by the compiler after allocation of all other memory space except the string buffer (if used). This buffer is only allocated for programs that use input/output.

NOTE: A special form of all input/output statements designed for buffer direct input/output using the special string variable BUF$ is described in the STRING PROCESSING section.

### Input Statement

Syntax: INPUT var , . . . , var

Description: This statement causes code to be generated which prints a $ prompt and space on the terminal device, then reads characters into the input buffer until 128 characters have been read or a carriage return symbol is read. A carriage return/line feed is printed when the last character in input.

At run-time, entry of a CONTROL X will print *DEL* and CR/LF and reset the buffer. A CONTROL O will backspace in the buffer and echo the deleted characters.

The variables specified var may be numeric or string, subscripted or simple type. When the program is "looking for" a number from the current position in the input buffer, it will skip leading spaces, if any and read a minus sign (if any) and up to five number characters. The numeric field is terminated by a space, comma, or end of line. If a non-digit character is read, or any other illegal condition a value of zero will be returned for the number.

If a string-type field is being processed, characters from the current position will be accepted including blanks until the variable field is terminated by a comma or end of line, or when it is "full". If no characters are available, a null string will be returned.

Examples:

```
INPUT  A,B,S$,B$
INPUT  A(N+!,M-1),B,A(4,N)
INPUT  A$(N),B$(N+1),D$
INPUT  B
```

Print Statement

Syntax: PRINT  out spec   delimiter   . . . delimiter  out spec

Description: This statement processes the list of out spec 's and puts the appropriate characters in the buffer. The buffer is then output to the terminal device.

An out spec may be a string expression or a numeric expression, or the output function TAB expr which inserts spaces in the buffer until the position expr is reached. Each item in the list is seperated by a delimiter which is a comma or semicolon. The buffer is divided into sixteen 8-character zones, which are effectively tab stops every eighth position. If a comma is used as a delimiter, the next item will begin at the first position of the next zone. If a semicolon is used, NO spacing will occur. A semicolon at the end of a Print statement will inhibit printing of a carriage return/line feed at the end of the line.

Examples:

```
PRINT A,B;C
PRINT A$(N);A$(N+!)
PRINT A,A$,B,B$
PRINT TAB(N=1),Z4
PRINT A;B;C;
PRINT A$;TAB(N+M);B$
```

# A/BASIC DISK INPUT/OUTPUT OPERATIONS

A/BASIC uses the facilities of the host operating system for input and output to sequential disk files. In order for programs generated by A/BASIC to operate properly when disk functions are used, the memory resident facilites of the DOS must be present.

Disk I/O in A/BASIC is channel-oriented meaning a file to be used for input or output is "opened" and assigned a channel number by which all further operations are performed. A/BASIC supports up to 10 channels which is the maximum number of files that may be open at any time.

All disk file names are defined identically to the operating system's file name conventions, i.e. the file name specifications are identical. With the exception of the CHAIN statement, all files used by A/BASIC are ASCII text files.

Many of the A/BASIC disk operations use the same DOS subroutines as would be used by asssembly-language programs so information as to disk operations listed in the disk system manufacturer's software manual will generally apply.

Note: In the descriptions of disk statements that follow the term "filnum" refers to a channel number which is a constant which may range from 0 to 9.

## OPEN FILES Statement
## CLOSE FILES Statement

These statements are used to initialize and close the DOS disk manager, respectively. The OPEN FILES statement initializes the DOS and prepares its internal values for disk input/output operations. Consequently, this statement must be executed before any files are opened, closed or operated on. This statement should only be executed once. The CLOSE FILES statement should be executed after all disk operations are complete. It calls the DOS to "clean up" after disk operations and will close any files that may still be open.

## OPEN Statement

SYNTAX: OPEN #filnum,strexp

This statement is used to open (initialize) a file for input or output, and assign the file a channel number. One and only one file may be open on a particular channel at a time. The DOS requires that a particular file be open for either read or write, not both. The OPEN statement will first search the file directory on the specified drive and open the file for read if a file by the name specified is found. If no such file is found, a file with the name specified will be opened (created) for write operation. The STATUS function may be used after a file is opened to determine its read or write status if necessary.

If an error occured during the open operation, the line specified by the last ON ERROR GOTO statement to be executed will be transferred to. The ERR funtion may be used to determine the error type.

An OPEN statement must be used prior to any I/O to the particular channel. Alsonote that it is the only time that the file name is explicity used. All futher references to the particular file as long as it is open is by means of the channel number.

The channel number must be a constant integer from 0 to 9. The file name specification is a string constant, variable or expression and must conform to the DOS requirements for legal file name specifications (drive, name, extension, etc.)

## OPEN Statement (Cont'd)

Below are some examples of legal usage of the OPEN statement.

| | |
|---|---|
| `OPEN #0,"TEST"` | Opens the file TEST and assigns to channel 0 |
| `A$="MASTER"`<br>`OPEN #2,A$` | Opens the file MASTER on channel 2 |
| `A$="MASTER"`<br>`OPEN #2,"1:"+A$+"TXT"` | Opens the file MASTER.TXT on dirve #1 and assigns it channel 2 (string expression evaluates to: 1:MASTER.TXT). |

## Close Statement

SYNTAX: CLOSE #filnum

This statement is used to close a file after I/O operations are complete and releases the channel. The file on the channel is closed for either read or write automatically. The channel is then available for reassignment to another file if desired.

Examples of legal usage:

```
CLOSE #2
```

NOTE: DO NOT CONFUSE THE OPEN FILES AND CLOSE FILES statements with the OPEN and CLOSE STATEMENTS AS THEY PERFORM ENTIRELY DIFFERENT FUNCTIONS.

## WRITE Statement

Syntax: `WRITE #filnum,outlist`

The WRITE statement causes a record containing the data specified to be written on the disk file as a single record. The file must be open for use as a write file (see OPEN).

The output list is a sequence of string or numeric constants, variables or expressions separated by commas. Each element in the list is considered to be an "item" within the disk record to be written. Strings and numeric values are written in ASCII form. Type conversion is automatic for numbers. The resulting record length must be less than or equal to a total of 128 bytes.

Examples of legal usage:

```
WRITE #3,A,B,C
WRITE #6,"DATA",N$,C$,Z$,T
WRITE #6,400+Z,A$+MID$(B$,4,M),M/2
WRITE #2,"MASTER FILE NUMBER "+STR$(N)
```

Results of writing a disk record:

Each item is written on the record with a comma item seperator. The records are variable length and use a carriage return character as an end of record terminator. The statement (assume variable N=10):

```
WRITE #2,25,-400,"WORD",N
```

produces a disk record which has the following format ( in hex ):

```
32  35  2C  2D  34  30  30  2C  57  4F  52  44  2C  31  30  0D
 2   5   ,   -   4   0   0   ,   W   O   R   D   ,   1   0  EOR
```

## READ Statement

SYNTAX: `READ #filnum,varlist`

The read statement causes the next record of the read file on the channel specified to be read into the A/BASIC I/O buffer. After the record is read, data items corresponding to the items in the variable list will be taken in order and stored in the appropriate variable locations. The variable list may include numeric or string type which may use subscripted variables.

The number of items given in the variable list should agree with the number of items on the disk data record (except as noted below). The file must be open for read and must be an ASCII text-type file. Disk records are variable length.

Rules for reading different data types: the following rules apply to and define the result of reading items from a disk record under various circumstances.

| NEXT VARIABLE IN | NEXT ITEM IN BUFFER (RECORD) IS .. | | |
|---|---|---|---|
| LIST IS ... | NUMERIC | STRING | EMPTY |
| NUMERIC TYPE | NUMBER | NUMBER if string contains legal decimal chars. | ERROR |
| STRING TYPE | STRING OF DIGITS | STRING | NULL STRING |

If there are more items on the record than variables in the list, they will be ignored. Note that numeric variables are not stored as binary bytes, rather they are converted to ASCII character representation before being written and converted to binary after being read.

## CHAIN Statement

Syntax: CHAIN strexp

This statement allows the BASIC program to load and begin execution of another (or an additional) program. The program name is specifed by the string expression.

The program to be CHAINed must have the following characteristics:

1) It MUST have been saved on the disk as a binary format program. A/BASIC produces HEX format object programs so A/BASIC-produced programs must be loaded and then saved in binary before being CHAINed.
2) A transfer address (starting address) must have been specifed when the binary program was saved.

Warning: The CHAIN statement uses the DOS's RUN function so if an error occurs (file not found, etc.) control will pass back to DOS, not the BASIC program.

The CHAIN function may also be used to load blocks of binary data, character data, or additional program if the following "trick" is used. When the additional data or program is saved, specify a transfer address that corresponds to the actual address of the BASIC statement following the CHAIN statement that will cause the data/program to be loaded.

Examples of usage:

```
CHAIN "1:UPDATE.BIN"
CHAIN "PROG7"

A$="2:QUERY.BIN"
CHAIN A$
```

## RESTORE Statement

## SCRATCH Statement

```
SYNTAX:  RESTORE #filnum, ... ,#filnum
         SCRATCH #filnum, ... ,#filnum
```

These statements are used to close and then reopen files on the specified channel(s). The channels must have been previously OPENed.

The RESTORE statement closes a file open for either read or write and opens it again for read. This is analogous to a "rewind" operation.

The SCRATCH statement closes a file open for either read or write and opens it for write.
WARNING: The SCRATCH statement destroys the file being SCRATCHed!!! It in effect deletes (destroys) the file and opens a new file with the same name. When a file is SCRATCHed all previous data is lost.

Because of the sequential nature of files supported by the DOS, these statements are used to "reposition" the internal "pointer" to the next record to read from or written to.

Examples of legal usage:

```
SCRATCH #2,#8
RESTORE #3
```

## KILL Statement

```
SYNTAX:   KILL stringexpr
```

The file name specified is PERMANENTLY DELETED from the system. Use with great care as the file may not be recovered.

Example:

```
KILL "TEMP3"
KILL A$
```

## DISK FUNCTIONS

The following fuctions are available for use with disk input/output operations. All operate as the other BASIC functions and return a numeric-type result.

### EOF Function   Syntax: EOF(#filnum)

Returns a value of 1 if an end of file condition exists on the file assigned to the channel specified, otherwise returns 0.
Example: `IF EOF(#4) = 1 THEN 550`

### FREE Function   Syntax: FREE(drive number)

Returns the number of sectors that are available (not used by existing files) on the disk drive unit specified.

### FILSIZ Function   Syntax: FILSIZ(#filnum)

This function returns the current length of the file specified in sectors.

Example: `IF FILSIZ(#4) = 35 THEN 440`

### STATUS Function   Syntax: STATUS(#filnum)

Returns the current status of the file specified as follows:

0 = File is not currently open
1 = File is open for read
2 = File is open for write

Examples:

```
ON STATUS(#8) GOTO 470,120

IF STATUS(#1) = 1 THEN 610

N2=STATUS(#5)
```

## USER-DEFINED STATEMENTS

A/BASIC V2.0 has provisions for the user to add up to three new user-defined statements. The keywords may be selected by the user and become part of A/BASIC's "vocabulary".

Operationally, the user-defined statements will generate jump-to-subroutine (JSR) hardware instructions when a user-defined keyword is processed. The JSR is to a fixed, user-supplied address which is assumed to be the entry point of a run-time subroutine resident in the system at run-time (i.e. the user must supply a "run-time package" for these functions). The user-defined statements may optionally include an arithmetic expression which is evaluated by the compiler and passed to the user routines in the accumulators.

To add user-defined statements to A/BASIC V2.0, the user must insert parameters into the compiler at the addresses shown in the table below. The first parameter is the keyword name, which must be exactly four characters long and is patched into the compiler beginning at the address given below. The second is the run-time user subroutine address. The third, optional parameter is whether or not A/BASIC should "look for" an arithmetic expression following the user keyword which is to be evaluated and passed to the subroutine (an expression may be a constant, variable or expression).

| | KEYWORD STRING ADDRESS | SUBR. ADDR M.S. BYTE | SUBR. ADDR L.S. BYTE | EXPR OPTION ADDR. |
|---|---|---|---|---|
| 1 | 05F0 - 05F3 | 0BEA | 0BEC | 0BE7 - 0BE8 |
| 2 | 05F5 - 05F8 | 0BF4 | 0BF6 | 0BF1 - 0BF2 |
| 3 | 05FA - 05FD | 0BFE | 0C00 | 0BFB - 0BFC |

To add the expression option, change the addresses shown from 03, 68 to 12, 38.

EXAMPLE: To add the statement PLOT (expr) to A/BASIC when the user routine has a run-time address of $3500 the following changes are made. 1. The ASCII values for the characters P,L,O,T are inserted from 05F0 to 05F3. 2. The address is inserted: $35 at addr 0BEA and $00 at addr 0BEC. 3. The expression evaluation option is activated by changing 0BDA and 0BDB to $12 and $38.

## USER-DEFINED STATEMENTS, CONT'D

After the changes have been correctly made, the compiler will accept the new statement PLOT. For example it may be used as:

```
200 PLOT N+1
```

for which the compiler will generate instructions:

```
LDAB  (addr of L.S. byte of N)      GET N
LDAA  (addr of M.S. byte of N)
ADDB #1                             ADD 1
ADCA #0
JSR $3500                           CALL USER SUBR.
```

Because the compiler "expects" an expression to follow the new keyword an error will result if some arithmetic value does not follow. If the new keyword was added without the expression option activated, it would have to be used alone such as:

```
200 PLOT
```

which would directly translate to the machine instruction

```
JSR $3500
```

## COMPILATION PROCEDURES

The first step in producing an A/BASIC program is to use the system's text editor to create the source program containing the A/BASIC program. A/BASIC accepts the file formats used by most editors.

The compiler is then called as a system command such as:

```
ABASIC,PROG1,OBJ1
```

The format above specifies the file PROG as being the source file and OBJ1 as the file on which the object is to be written. The object file specification may be ommitted for a "listing only" compile.

A/BASIC is then loaded by the DOS and begins compilation. During the second pass a formatted listing is produced which will include any error messages (see COMPILE TIME ERRORS). The listing usually consists of three fields: the first is the hex address where the object code starts for the BASIC source line; the second field is the statement line number if any; and the last field is the BASIC source line.

After compilation is complete, the program statistics, load map and symbol table (if the program included an OPT S statement) are printed. The load map lists the names and addresses (main entry point) of the subroutine packages the compiler selected and included in a particular program. The BASLIB module names, functions and compiler addresses are:

| NAME | FUNCTION | SIZE | ADDRESS |
|---|---|---|---|
| OUT* | TERMINAL OUTPUT MODULE (PRINT) | $00E3 | $1D00 |
| INP* | TERNIMAL INPUT MODULE (INPUT, READ) | $0125 | $1DF3 |
| DSK* | DISK I/O INTERFACE (DISK OPS.) | $00C2 | $1F20 |
| MUL | MULTIPLY (ARITHMETIC) | $0080 | $2000 |
| DIV | DIVIDE (ARITHMETIC) | $007E | $2080 |
| RND | RANDOM NUMBER GENERATOR (RND) | $001A | $20FE |
| ST1* | STRING PRIMITIVES AND FUNCTIONS | $00E5 | $2120 |
| ST2* | STRING FUNCTIONS (SUBSTR,VAL,STR) | $0079 | $2205 |
| ARR | ARRAY MULTIPLY | $0015 | $08ED |
| IND | ARRAY INDIRECT LOAD | $000A | (gen.) |

* These modules have multiple entry points.

## COMPILER OPERATION, CONT'D

The object file output is in hex "Mikbug" format and may be loaded into memory using the system's hex loader. The program starting address is $1000 by default and will be different if an ORG statement was processed by the compiler before any other executable statements.

### Error Messages and Processing

When A/BASIC detects an error in the source program during either the first or second pass it will print the source line in error and a message with a error code (the codes are listed in the appendix). The line below the erroneous source line will have an up arrow showing the approximate position of the error. This error location is about 95% accurate.

When an error is detected on a source line the compiler will not process the line further even if it is a multiple statement line, so the rest of the source line should be examined carefully for possible undetected errors.

# A/BASIC V2.0 MEMORY UTILIZATION MAP

| | |
|---|---|
| 3FFF | Additional 4K available for table expansion |
| 2FFF* | SYMBOL TABLE - uses 6 bytes/entry<br>100 symbols in 12K system |
| 2DA7* | LINE REFERENCE TABLE - 4 byte/entry<br>605 in 12K system |
| 2432 | BUFFERS and STACKS<br>Input and Output Buffers<br>Parser Stack<br>For/Next Stack<br>Machine Stack |
| 227F | BASIC SUBROUTINE LIBRARY AREA<br>(BASLIB2) |
| 1EFF | COMPILE-TIME I/O PACKAGE |
| 1B7F | COMPILER MAIN PROGRAM<br>(I/O Jump Table 0100 - 0114) |
| 00FF | COMPILER WORKING STORAGE |
| 001F | RESERVED FOR OPERATING SYSTEMS |
| 0000 | |

* May be user-modified to expand tables

## COMPILER INTERNAL MEMORY ADDRESSES

Some important internal addresses are listed below along with other realted information. The source to run-time and compile-time I/O routines are provided with the compiler in assembler source format. Any modifications to these routines are at the user's risk. I/O routines that are user-modified must retain identical functions, entry points and size as the original routines provided.

| Address | Definition |
|---|---|
| 0022-0023 | Data address pointer - var. storage allocation pointer |
| 0024-0025 | Object address pointer - object code address pointer |
| 0030 | Pass flag - 0=pass1 |
| 003A | Listing page number |
| 003B | Listing line counter |
| 009B | Object code buffer byte count |
| 009C-009D | Object code buffer load address |
| 009E-00B5 | Object code buffer (binary format) |
| 012A-012B | End addr of line # table |
| 0131-0132 | End addr of symbol table |
| 014F-0150 | Default initial value for data pointer |
| 015F-0160 | Default initial value for object code pointer |
| 03E2 | Number of lines per page minus nine |
| 1C96 | # of nulls at end of line (min. 1) - compile time |
| 1C9F-1CA0 | Address of listing output character routine |
| 1D38-1D39 | Run time char. out routine address (PRINT) |
| 1E9D-1EBE | " " " " " " (INPUT) |
| 1EAC-1ECD | " " " input " " " |
| 1EAF | " " backspace character |
| 1EBF | " " delete line character |
| 2388-23D8 | Source input buffer |
| 23D9-2432 | Listing output buffer |

## A/BASIC Run-Time Environment

### Memory Assignments:

A/BASIC programs should reserve addresses below $0030 for the operating system and for BASLIB temporary storage. The usage of this area of memory by A/BASIC is listed below:

| | |
|---|---|
| $0000-$000F | Reserved for operating system |
| $0010-$0011 | File control block temp. addr. (DSK) |
| $0012-$0013 | Error trap address (ST2,DSK,MUL,DIV,INP) |
| $0014 | Error type code (ST2,DSK,MUL,DIV,INP) |
| $0015-$001F | Not used but reserved for future use |
| $0020-$0021 | XR temporary (INP,OUT,ST1,ST2) |
| $0022-$0023 | I/O Buffer pointer (INP,OUT,DSK,ST2) |
| $0024-$0025 | Reserved |
| $0026 | I/O Buffer zone counter (INP,OUT,DSK,ST2) |
| $0027-$0028 | String Buffer pointer (ST1,ST2) |
| $0029-$002A | String XR temp (ST1,ST2) |
| $002B-$002C | String temp (ST1,ST2)<br>Multiply overflow high-order 16 bits (MUL) |
| $002D-$002F | String functions temp. (ST1,ST2) |

## Operating System and Monitor Interfaces

If an A/BASIC program uses any of the disk I/O operations the host DOS must be resident in memory as A/BASIC will call its routines for I/O.

If the program uses INPUT or PRINT statements, the program will call the system ROM monitor for basic character I/O subroutines:

INPUT CHARACTER  ADDRESS=$E1AC
OUTPUT CHARACTER  ADDRESS=$E1D1

References to the above routines in BASLIB are made using these monitor subroutines only.

If the program uses any of the REAL-TIME statements TASK ON/OFF, SWITCH, ON IRQ GOTO, ON NMI GOTO, etc. the system monitor must be Microware's RT/68MX or RT68MXP.

## A/BASIC V2.0 LANGUAGE SUMMARY

### ASSIGNMENT:

LET
POKE

### CONTROL:

CALL
GOSUB
ON ERROR GOTO
ON/GOSUB
FOR/TO/STEP
RETURN
ON OVR GOTO
STOP
GOTO
IF/THEN
ON NOVR GOTO
NEXT
IF/GOSUB
ON/GOTO

### INPUT/OUTPUT:

INPUT
OPEN
CHAIN
PRINT
CLOSE
RESTORE
OPEN FILES
READ
KILL
CLOSE FILES
WRITE
SCRATCH

### SYSTEM CONTROL AND REAL-TIME:

GEN
ON NMI GOTO
TASK .. ON
IRQ ON
RETI
TASK .. OFF
IRO OFF
STACK
ON IRQ GOTO
SWITCH

### COMPILER DIRECTIVES:

BASE
OPT
ORG
REM
END
DIM

### NUMERIC FUNCTIONS:

ABS
PEEK
SUBSTR
FREE
POS
TAB
VAL
FILSIZ
CLK
ASC
ERR
STATUS
RND
LEN
EOF
SWAP

### STRING FUNCTIONS:

CHR$
STR$
LEFT$
TRM$
RIGHT$
BUF$
MID$

### OPERATORS:

+ ADD
- SUBTRACT
/ DIVIDE
* MULTIPLY
- NEGATIVE

& AND
! OR
% EXCL. OR
# NOT

+ CONCATENATE (STRING)

# DIFFERENCES BETWEEN VERSION 1.0 (CASSETTE) AND 2.0 (DISK)

## Additions to 2.0

### New Statement Types Added:

OPEN CLOSE READ WRITE
CHAIN RESTORE SCRATCH KILL
(User Defined)

### New Functions Added:

ERR EOF FREE FILSIZ
STATUS SWAP

### Statement Types Deleted:

TREAD TWRITE

### Statements Modified

ON ERROR GOTO - Now sets/removes error trap across multiple lines
DIM - permits individual sizing of string variables
POKE, PEEK - accepts expressions for addresses
STOP - jumps to DOS instead of RT68
I/O STATEMENTS - No longer RT/68 dependent

### Compile-time and Source File Differences:

Line numbers optional
Generates load map at end of compilation
Formatted listing
Monitor-independent I/O

### Run-time Differences:

Individually sized strings
Monitor-independent I/O
Full error trap capability
BASLIB Subroutine elements improved and smaller
Larger run-time scratch area
No longer supports tape I/O

# A/BASIC V2.0 COMPILE-TIME ERROR CODES

02 Line number duplicated or out of sequence
03 Unrecognized statement
04 Syntax error
05 Variable name missing or in error
06 Equal sign missing
07 Undefined line reference: GOTO or GOSUB to nonexistant line number
08 Right parenthesis missing or misnested
09 Operand missing in expression
10 Destination line number missing or in error
11 Number missing
12 Misnested FOR/NEXT loop(s)
13 Symbol table overflow*
14 Illegal task number - must be 0 to 15
15 Missing or illegal usage of relational operator(s)
16 delimiter ( , or ; ) missing
17 Quote missing at end of string
18 Illegal type or missing counter variable in FOR statement
19 Redefined array
20 Error in array specification: subscript missing; too many subscripts
21 Error in array specification: subscript zero or larger than 255
22 Variable storage overflow, tried to allocate past $FFFF
23 Reference to undeclared array
24 Subscript error
25 Missing, illegal type or incorrect function argument (numeric)
26 Illegal option
27 Unrecognized operator in string expression
28 Concatenation operator (+) missing
29 Missing, illegal type or incorrect function argument (string)
30 Too many FOR/NEXT loops active - max is 16
31 Line reference table overflow*
32 Program storage overflow - tried to allocate past $FFFF
33 ON or OFF keywords missing in IRQ statement
34 GOTO or GOSUB missing
35 Illegal channel number: must be 0 to 9
36 Error in disk I/O list

* These error types are not program errors. If more system memory is available the tables may be expanded to include more entries.

## A/BASIC V2.0 RUN-TIME ERROR CODES

The ERR function will return on of the following codes after an error occurs at run-time:

ERRORS 0 to $31_{10}$ - DISK ERRORS The codes used are identical to those used by the host DOS and may be found in the system's DOS manual. All codes may not be implemented by DOS.

ERROR 32 - MULTIPLY OVERFLOW The result of a multiplication exceeds the range +32767 to -32768. The result was the low order 16 bits of the result and the high order 16 bytes are saved in the fast scratch area.

ERROR 33 - DIVIDE ERROR A divide with a zero divisor was attempted. A result of zero was returned.

ERROR 34 - CONVERSION ERROR The BASLIB ASCII-to-binary conversion routine read illegal, oversize or no input. A value of zero was returned.

## A/BASIC GLOSSARY

The terms defined below may be unfamiliar to some programmers or used in a special context in this manual.

ALLOCATION: The process of assignment of a specific memory address to variables or machine instructions.

CODE GENERATION: The process of creating machine language instructions.

COMPILE-TIME: Used to describe the time during which the BASIC program is being processed by the compiler.

LIBRARY: A collection of subroutines within the compiler (BASLIB) which are used to generate subroutines within the machine language program. It includes "images" of subroutines for mathematical functions, string processing, input/output, array operations, etc.

LINE REFERENCE: A reference from a statement to a line number which is that of another line.

LINE REFERENCE TABLE: A table which is kept by the compiler which is used to store the correspondences between referenced line and the memory address assigned to the line.

LINKER/EDITOR: A portion of the A/BASIC compiler program which places subroutines which are required in the object program. The images are obtained from the library and the linker/editor inserts absolute addresses of the subroutines in the program. Only one copy of a subroutine will ever be generated in a program, and only those that are required by that specific program.

OBJECT CODE: The machine-language instructions generated by the compiler.

OBJECT FILE: The tape or other media file that the compiler has written the machine language program on.

OBJECT PROGRAM: The machine language program produced by the compiler from the BASIC source program.

PASS: A complete scan of the source program. A/BASIC is a twopass compiler so it must completely read the source program twice.

REAL-TIME EXECUTIVE: The portion of the RT/68 Operating System that schedules and controls task execution.

RUN-TIME: Used to descirbe events or the time during which the machine language program is being executed.

SOURCE FILE: The tape or other media containing the A/BASIC program which is to be compiled.

SOURCE PROGRAM: The BASIC program text to be compiled.

SYMBOL TABLE: A table kept by the compiler during compilation that contains information about the correspondence between variable names and assigned memory addresses, and type of variable.

SYNTAX: Rules for proper construction of parts of BASIC statements.

TWO'S COMPLIMENT: A method of binary representation of numbers where negative numbers are represented as the result of subtracting the absolute value of the number from zero.